DCast: Sustaining Collaboration in Overlay Multicast despite Rational Collusion

Haifeng Yu
National University of Singapore
Republic of Singapore
haifeng@comp.nus.edu.sg

Phillip B. Gibbons
Intel Labs
Pittsburgh, PA, USA
phillip.b.gibbons@intel.com

Chenwei Shi
Mozat Pte Ltd
Republic of Singapore
shichenweixjtu@gmail.com

ABSTRACT

A key challenge in large-scale collaborative distributed systems is to properly incentivize the rational/selfish users so that they will properly collaborate. Within such a context, this paper focuses on designing incentive mechanisms for overlay multicast systems. A key limitation shared by existing proposals on the problem is that they are no longer able to provide proper incentives and thus will collapse when rational users collude or launch sybil attacks.

This work explicitly aims to properly sustain collaboration despite collusion and sybil attacks by rational users. To this end, we propose a new decentralized *DCast* multicast protocol that uses a novel mechanism with *debt-links* and circulating debts. We formally prove that the protocol offers a novel concept of *safety-net guarantee*: A user running the protocol will always obtain a reasonably good utility despite the deviation of *any* number of rational users that potentially collude or launch sybil attacks. Our prototyping as well as simulation demonstrates the feasibility and safety-net guarantee of our design in practice.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

Keywords

Algorithmic mechanism design, incentive mechanism, rational collusion, sybil attack, whitewashing attack, overlay multicast

1. INTRODUCTION

The past decade witnessed the emergence of many large-scale collaborative distributed systems, where the individual rational (selfish) peers are supposed to collaborate. How to incentivize these rational peers to sustain such collaboration (and avoid the *tragedy of the commons*) is a key and well-known problem [12, 20, 21, 22]. This paper focuses on one particular kind of collaborative distributed system, *overlay multicast*, for its practical importance. Overlay multicast has key practical applications such as video streaming of sporting events or TV programs. A peer in overlay multicast is supposed to forward/relay the multicast data to other peers. Without proper incentives however, a rational peer may choose to save bandwidth by not forwarding the data to others. Such user preference of minimizing bandwidth consumption has been widely observed in previous measurement studies [1, 24], where 30% of the Napster users deliberately under-reported their available bandwidth to discourage others from downloading from them and 70% of the Gnutella users did not contribute any bandwidth. One way to sidestep this incentive problem in overlay multicast, of course, is to deploy servers with high outgoing bandwidth to directly send the data to each individual peer. Unfortunately, compared to other collaborative systems, multicast is much less amenable to such cloud computing style of solution, due to its excessive bandwidth requirements. As a result, current large-scale commercial multicast systems (e.g., Adobe Flash Player 10.1 and PPLive online TV platform [22]) often rely on overlay multicast.

To prevent peers from free-riding, researchers have proposed a number of interesting and practical proposals on incentivizing overlay multicast in particular [9, 14, 15, 18, 22] and various collaborative systems in general [2, 26, 30]. Unfortunately, these prior solutions all share the key limitation that they can no longer properly provide incentives if rational peers collude, launch *sybil attacks*[1] [4], or launch *whitewashing attacks*[2]. In particular, some of these incentive mechanisms may fail even if each rational peer only colludes with a few other peers. While in some other designs, a peer may survive purely on "seed money/credit" obtained via repeated whitewashing (see Section 2). Thus in the presence of collusion or sybil/whitewashing attacks, these prior proposals will again lead to the tragedy of the commons, where no peer can obtain any data from other peers. It is worth noting that collusion in the virtual world is far easier than one might expect at first thought. For example, all peers using the same "hacked" version of a protocol (e.g., posted online) are already coordinated and can readily collude. The colluding peers do not need to know each other beforehand and the collusion can form *on-the-fly*.

Challenges in defending against collusion. The inability of these previous approaches to deal with collusion is related to the following two challenges. First, the key to incentivizing collaboration is always the implementation of a (potentially implicit) punishment mechanism, to punish those peers who fail to collaborate. The presence of collusion makes it challenging to punish. Evicting a peer is no longer an effective punishment — the evicted peer may obtain multicast data from its colluding peers. This problem is further

[1]In this paper, we focus on *rational sybil attacks*, in which it profits a user to create many identities.

[2]*Whitewashing attack* refers to a user abandoning her/his identity to evade punishment and then rejoining with a new identity.

complicated by sybil attacks and whitewashing attacks. Existing sybil defense mechanisms (e.g., [28]) need to assume collaboration among the peers in the first place.

Second, in some cases the colluding peers might be able to obtain the multicast data from each other more efficiently. For example, suppose the multicast protocol is based on random gossiping, for better robustness against churn. If the colluding peers have low churn, then they can switch to using more efficient tree-based multicast amongst themselves. Such deviation is already profitable. Furthermore, the colluding peers can either continue to gossip with the non-deviators as usual, or they can do so less frequently. Detecting such deviation from the non-deviators' perspective is challenging, because occasional participation is indistinguishable from legitimate but slow participation.

Our results: Safety-net guarantee and the DCast protocol. This work aims to properly sustain collaboration in overlay multicast despite collusion and sybil/whitewashing attacks by rational users. As hinted by the previous example, we will first show that in overlay multicast, it is *impossible* in practice to prevent profitable deviations by a colluding set of peers. This contrasts sharply with previous proposals that directly aim at eliminating profitable deviations (since they do not deal with collusion) [9, 14, 15, 18, 22]. On the other hand, note that it is not deviation that is harmful — rather, it is the deviation's negative impact on other (non-deviating) peers that is harmful. This leads to our novel concept of a *safety-net guarantee* in a game theoretic context, which formalizes the goal of this work. Intuitively, a protocol offers a *safety-net guarantee* if a peer running the protocol (called a *non-deviator*) is guaranteed to at least obtain a reasonably good utility (called the *safety-net utility*), despite deviations by *any number* of rational users who may collude or launch sybil/whitewashing attacks. Note that in classic security settings focusing on malicious attackers, one can often assume that the number of malicious users is limited. In comparison in our game theory context, since all users are rational and aim to maximize their utilities, *all* users are ready to collude or deviate if opportunities arise.

While the concept of a safety-net guarantee helps to circumvent the above impossibility, the two fundamental challenges discussed earlier remain. This paper then proposes a novel *DCast* multicast protocol, which is the first practical overlay multicast protocol with such a safety-net guarantee. In contrast, previous designs [9, 14, 15, 18, 22] lack a safety-net guarantee, and a non-deviator can fail to obtain any multicast data at all from other peers in the presence of collusion among rational peers. DCast is also robust against a small number of malicious peers. For achieving these guarantees, DCast requires no crypto operations except for basic message authentication and encryption.

Formally, we prove that DCast offers a safety-net guarantee with a good safety-net utility. We further implement a DCast prototype in Java, as well as a detailed simulator for DCast. Experimental results from running the prototype on Emulab (with 180 peers) and from simulation (with 10,000 peers) confirm the feasibility and safety-net guarantee of our DCast design in practice.

Key techniques. DCast achieves the safety-net guarantee via the novel design of *debt-links* and *doins*. Debt-links instantiate the idea of *pairwise entry fees*, which allow a peer to interact with some specific peers to a limited extent. A debt-link from a peer A to a peer B is established by B sending some junk bits to A. Doins (shorthand for *debt coins*) instantiate the idea of *profitable interactions*. Doins are circulating debts and can be viewed as a variant of bankless virtual currency. A doin can be issued by any peer and circulates only on debt-links. A doin *occupies* a debt-link that it passes through, until the doin is eventually paid.

Occupied debt-links serve as an effective punishment, even in the presence of collusion, to a peer that fails to pay for a doin. The doin payment amount is explicitly designed to be strictly larger than the cost of issuing the doin. Thus with proper debt-link reuse, the accumulated profit from doin payments will offset and further exceed the cost of accepting a debt-link establishment. This in turn incentivizes peers to accept debt-link establishments and issue doins. Under proper parameters, such profit also conveniently incentivizes the colluding peers, even if they can enjoy a lower cost of disseminating data among themselves.

Summary of contributions. In summary, this paper aims to incentivize overlay multicast in the presence of rational collusion, a setting where prior solutions will collapse. We make the following main contributions: i) we introduce the notion of a safety-net guarantee in a game theoretic context, ii) we present the novel DCast multicast protocol, iii) we formally prove that DCast offers a safety-net guarantee, and iv) we demonstrate via prototyping and simulation the feasibility and safety-net guarantee of the design in practice.

2. RELATED WORK

Concepts related to our safety-net guarantee. Designing algorithms for sustaining collaboration among rational users is usually referred to as algorithmic mechanism design [5]. Almost all previous efforts in algorithmic mechanism design aim to eliminate profitable individual deviations (by forming a Nash equilibrium), or to eliminate profitable group deviations (by forming a collusion-resistant Nash equilibrium [19]). In comparison, this work shows that such equilibrium is not possible in our context and thus explicitly does not focus on equilibrium. Instead, we aim to protect the utility of the non-deviators. Our goal is more related to the *price of collusion* [8], which quantifies the negative impact of collusion on the *overall* social utility in a congestion game. In comparison, our safety-net guarantee bounds the negative impact of collusion on the utility of *individual* non-deviators in a multicast game. Furthermore, we consider all pareto-optimal strategy profiles of the colluding peers, while the price of collusion considers only the strategy profile that maximizes the overall sum of the utilities of the colluding peers.

Incentivizing overlay multicast. A number of interesting and practical techniques have been developed for building incentives into overlay multicast [9, 14, 15, 18, 22]. When rational users collude, however, none of these approaches can continue to provide proper incentives.

Specifically in the Contracts system [22], a peer issues *receipts* to those peers who send it data. These receipts serve to testify those peers' contribution. Contracts is vulnerable to even just *two* colluding users. For example, a user A wanting the multicast data can trivially obtain fake receipts from a buddy B who does not actually need the multicast data. (In return, A may give fake receipts to B in some other multicast sessions where A does not need the data.) Receipt validation as in Contracts does not help here since B does not need the data. If we have another colluding peer C who further gives B fake receipts, then A's contribution will further have a good *effectiveness* (see [22] for definition), meaning that A is credited with contributing to other contributing peers. BAR gossip [15] and FlightPath [14] rely on evicting peers as an effective punishment. Section 1 already explained that this will not be effective with collusion. The approaches in [9, 18] punish a deviator by not sending it data, which is equivalent to eviction. Similarly, all these approaches are vulnerable to rational sybil attacks and whitewashing. Second, in scenarios where the colluding peers can obtain the data from each other more efficiently, the protocols in [9, 15, 18, 22] can no longer guarantee non-deviators' utility. FlightPath [14] achieves a stronger guarantee under the assumption that a peer will not deviate unless the deviation will bring at least ϵ (e.g., 10%) extra utility, which corresponds to an ϵ-Nash [16]. The value of ϵ usually is small since otherwise the ϵ-Nash assumption itself becomes questionable. However, the colluding peers may easily adopt optimizations specific to their own characteristics (e.g., low churn rate) that far exceed such 10% threshold. In contrast, the safety-net guarantee of DCast continues to hold even if the utility gain of the colluding peers is much higher (e.g., 100% or 200%).

In the grim trigger approach [13], if a peer does not receive enough multicast data, it conceptually signals the multicast root to shut down the entire multicast session. While this is indeed robust against rational collusion, the approach is rather vulnerable to even just a single malicious peer and to various performance instabilities in the system. More recently, Tran et al. [25] aims to maintain collusion-resilient reputation scores for peers, but their approach's guarantee is rather weak and colluding peers can increase their reputation scores unboundedly as the number of colluding peers increases. Finally, a preliminary version of this work was published as a 2-page Brief Announcement [29].

Incentivizing other collaborative systems. There have been many efforts on incentivizing general p2p systems. These efforts are largely heuristics and often do not even prevent individual deviation, let alone dealing with collusion. A further common limitation of these approaches [2, 11, 21] is the need to give new users some "seed money/credit" to bootstrap them (regardless of whether currency or credit is explicitly used). As acknowledged in [2], such "seed money/credit" necessarily introduces vulnerabilities to sybil and whitewashing attacks. In contrast, our DCast design manages to avoid "seed money/credit" completely. In the next, we will discuss in more detail efforts whose techniques are more related to ours.

Samsara [3] is a p2p backup system that aims to prevent freeriding. When a peer B wishes to back up data on peer A but A has nothing to back up on B, A will force B to store a *storage claim* of the same size as B's backup data. Such design does not actually sustain collaboration when users are rational — since A's utility decreases here, A actually has incentive *not* to engage in such an interaction.

Many previous efforts [6, 11, 17] use *credit chains* (including one-hop reputation [21] as length-2 credit chains) to achieve a similar functionality as circulating currency. In these designs, there is no mechanism to ensure that credits will be honored. DCast avoids such problem by allowing doins to circulate only on established debt-links.

KARMA [26] and MARCH [30] use a standard virtual currency system to sustain collaboration, where a peer makes (spends) money when offering (obtaining) service. Their key feature is to assign each peer A some random set of other peers as A's bank to maintain A's balance. Such design is clearly vulnerable to on-the-fly collusion between A and its bank peers. As explained in Section 1, since all peers are rational in a game theory context, all peers are ready to collude and deviate if opportunities arise. Thus such collusion can occur as long as it benefits the bank peers as well.

Finally, *bandwidth puzzles* [23] aim to address rational collusion in a general p2p context. In this approach, a trusted verifier first collects information regarding all data transfers among the peers, and then presents bandwidth puzzles to all peers *simultaneously* to verify that each peer indeed has the content that it claims to have. Despite its name, this approach still needs to assume that the computational power of each peer is roughly the same, otherwise one peer can help its colluding peers to solve their puzzles. In contrast, our solution does not need to rely on such a strong and often unrealistic assumption. Our design does not need to collect and verify global information about data transfers either.

3. SYSTEM MODEL AND SAFETY-NET GUARANTEE

Basic model. A *user* of the multicast system has one or more identities (i.e., our model allows sybil attacks), and each identity is called a *peer*. During the multicast session, each peer has a (fixed) IP address and a (fixed) locally generated public/private key pair. Peers are identified by their public keys, and their IP addresses serve only as a hint to find the peers. We do not bind the IP address to the public key and we allow IP address spoofing. The multicast *root* is the source of the (signed and erasure-coded) *multicast blocks*, which encode application-level data such as video frames. The root has limited outgoing bandwidth, and can only directly send some blocks to a small set of peers. Depending on the protocol, the root may only send a small number of blocks to each peer in this set, and the set membership may also change over time. The root has a publicly-known IP address and public key, and always follows our protocol. The peers and the root have loosely synchronized clocks with bounded clock errors.

Rational peers and utility. We assume that all peers are *rational*, except a small number of *malicious* (i.e., byzantine) peers. A rational peer is selfish and aims to maximize its *utility*. The utility increases with the number of multicast blocks (i.e., multicast bits) received, and decreases with the number of non-multicast bits received and the number of bits sent. To unify terminology, we call bits sent or non-multicast bits received as *cost bits*. Cost bits thus constitute an overhead that a rational peer wants to minimize.[3] Usually the benefit of receiving one multicast bit will far exceed the cost of sending/receiving one cost bit, since it is more important to get the multicast data than reducing the cost. Except the above, our discussion and proofs will not depend on the specific forms of the utility functions.

A rational peer is a *non-deviator* if it chooses to follow our protocol, otherwise it is a *deviator*. There is no *a priori* limit on the number of deviators; after all, *every* rational peer seeks to maximize its utility and hence may potentially deviate.

For each deviating peer v, we assume that there exists some "protocol gaming" constant σ_v such that *drawing on the collusion*, v is now (only) willing to incur one cost bit for every σ_v multicast bits received on expectation. Here due to the collusion, σ_v can be close to 1, or even above 1 if some other colluding peers are willing to sacrifice for v. Different deviating peers may have different σ_v values. Let σ be a constant such that $\sigma \geq \sigma_v$ for most v's. Having this σ constant will be sufficient for us to later reason about the guaranteed safety-net utility for non-deviators, since those (small number of) deviating peers whose σ_v exceeds σ can be no worse than malicious peers.

Pareto-optimal collusion strategies. We need to properly rule out irrational deviations when reasoning about rational peers, since otherwise they are no different from byzantine peers. We will apply the notion of *pareto-optimality* to do so. A *collusion strategy* defines the set of all the deviators, as well as the strategy adopted by each

[3]For example, doing so may save the power consumption and potential data charges of a mobile user. Even for home users with a flat-rate Internet service, reducing bandwidth consumption improves the experience of other concurrent users/applications at home. As discussed in Section 1, such user preference has been widely observed in previous measurements [1, 24].

of them. A collusion strategy α is *pareto-optimal* if there does not exist another collusion strategy β such that i) the sets of deviators in α and in β are the same, ii) each deviator has the same or increased utility under β compared to under α, and iii) at least one deviator has increased utility under β.

Because the colluding peers are rational, it suffices to consider only pareto-optimal collusion strategies since it is always better for them to switch from a non-pareto-optimal strategy to a corresponding pareto-optimal one. On the other hand, also note that there are many different pareto-optimal collusion strategies. Some of them may maximize the total utility across all the deviators. Others may maximize the utility of one specific deviator while sacrificing the utilities of all other deviators. One can also imagine a wide range of strategies in between. We do not make assumptions on which of these strategies will be adopted by the collusion — rather, our safety-net guarantee will hold under any pareto-optimal collusion strategy.

Novel concept of safety-net guarantee. As explained in the example from Section 1 regarding colluding peers with low churn, detecting (and preventing) certain deviations is rather difficult if not impossible in overlay multicast. Fundamentally, this is because the utility function in multicast involves performance overheads. Unless a protocol offers *optimal* performance for each possible subset of the peers (without knowing their specific properties such as low churn rate), some subset can always profit by deviating and switching to a more optimized protocol.

With such impossibility, we do not intend to prevent deviation. Rather, we aim to protect the utility of the non-deviators, by introducing the concept of *safety-net guarantee* in a game theoretic context. A safety-net guarantee requires that if a peer A chooses to stick to the protocol and if all peers are rational, then A should at least obtain a reasonably good utility (called the *safety-net utility*), despite *any* set of peers deviating from the protocol. Considering "any" set of peers is necessary here, since all users aim to maximize their utilities and thus all users are ready to collude or deviate if opportunities arise. We do not need to protect the deviators — if a deviator's utility is below the safety-net utility, it can always switch back to being a non-deviator.

The safety-net guarantee provides a lower bound on the utility of the non-deviators: When most peers do not deviate, the non-deviators' utility will likely be higher. One might argue that any loss of utility due to other peers deviating is unfair to the non-deviators. However, it would also be unfair to prevent colluding deviators from benefiting from advantageous factors such as low churn rate among themselves. In some sense, it is the non-deviators who prevented the system as a whole from using a more efficient protocol in such cases.

The safety-net guarantee by itself does not guarantee the utility for a non-deviator if malicious peers aim to bring down its utility. The need for this limitation is simple: Malicious peers can always send junk bits to a specific non-deviator, and drive down that non-deviator's utility arbitrarily. Dealing with this kind of targeted DoS attack is clearly beyond the scope of the safety-net guarantee. Note however that DCast offers additional properties beyond the safety-net guarantee, and we will explain later how DCast is robust against malicious users.

4. DCast DESIGN AND INTUITION

This section discusses the conceptual design and intuition of DCast, while leaving the protocol level details and formal proofs to Sections 5 and 6, respectively.

4.1 Design Space Exploration

We start by exploring the design space for effective punishment in the presence of collusion, which will naturally lead to the key design features in DCast.

Entry fee as effective punishment. A natural (and perhaps the simplest) punishment mechanism in the presence of collusion and sybil/whitewashing attacks is to impose an entry fee on each new peer. With proper entry fee, evicting a peer will constitute an effective punishment despite collusion. The key question, however, is how to design this entry fee and how to evict a peer when needed.

Since overlay multicast needs peers to contribute bandwidth, the entry fee must be in the form of bandwidth consumption. For example, if we instead use computational puzzles, then users with ample computational resources may still prefer whitewashing over contributing bandwidth. Using bandwidth consumption as the entry fee turns out to be tricky. Directly paying this entry fee to the root would overwhelm the root. Paying this entry fee to individual peers would enable colluding peers to accept fake entry fees from each other and vouch for each other. Furthermore, this entry fee must be in the form of sending junk data since a new peer has no useful data to send. This means that the individual peers have negative incentive to accept this entry fee.

Evicting a peer is also tricky under collusion due to the difficulty of disseminating the eviction information to all peers. The colluding peers clearly have incentive to interfere and stall the dissemination of an eviction notice.

Pairwise entry fees and profitable interactions. One way to overcome the above difficulties is to use *pairwise* bandwidth entry fees. Traditional *system-wide* entry fees admit a peer into the global system and entitle it to interact with all peers in any way it wants. With *pairwise* bandwidth (entry) fees, a peer sends junk data to some specific peers to be allowed to interact with only those peers and in a limited capacity that is proportional to the fee. Such pairwise nature prevents a colluding peer from giving other colluding peers interaction access to non-deviators. It also conveniently enables individual peers to unilaterally evict a peer, overcoming the previous difficulty of disseminating the eviction information globally.

The system still needs to properly incentivize individual peers to accept this (junk data) entry fee. One way to do so is to allow them to later *profit* from the interactions with those peers who paid the entry fee. Such profit hopefully can exceed the cost of accepting the entry fee. Rather conveniently, a properly designed profit here can further incentivize colluding peers to interact with non-deviators, even when the colluding peers may be able to obtain multicast blocks from each other more efficiently.

4.2 DCast Design

DCast essentially instantiates the above ideas of pairwise entry fees and profitable interactions.

Basic framework. Similar to several recent efforts [9, 14, 15] on incentivizing overlay multicast, DCast uses standard pull-based gossiping to disseminate multicast blocks for its simplicity and its robustness against churn. The multicast session is divided into *intervals*, and each interval has a fixed number of synchronous gossiping *rounds* (e.g., 30 rounds of 2 seconds long each). In each round, the root sends signed and erasure-coded multicast blocks to a small number of random peers, and all other peers each pull multicast blocks from another random peer. Before sending a block to a peer, the root requires the peer to send $\mathcal{D}_{\text{root}}$ (e.g., 3) *junk blocks* to the root, where each junk block is of the same size as a multicast block. To avoid delay, the sending of the junk blocks can be done before the round starts. (Note that these junk blocks are not "entry fees" and the number of junk blocks received by the root does not grow with system size.) Whenever a peer accumulates a sufficient number of erasure-coded multicast blocks for a given (typically earlier) round, a peer can decode the video frames for that round. If a peer fails to obtain enough blocks before a certain deadline (e.g., 20 rounds after the blocks were sent by the root), it will consider the frames as lost.

DCast allows peers to dynamically join and leave the multicast session, by contacting the root. The root maintains a list of the IP addresses of all peers. This list is obviously subject to sybil attack and we will address that later. To find other peers to gossip with, a peer may periodically request from the root a random *view* containing a small number of random IP addresses in the root's list. Note that a rational peer has no incentive to repeatedly request a view since it consumes its own bandwidth as well.

Incentive mechanism: Debt-links and doins. Rational colluding peers may profitably deviate from the above protocol in many ways. For example, a colluding peer A can pretend that it has no multicast blocks to offer when a non-deviator pulls from A. A can also pull from multiple non-deviators in each round. A user may further launch a sybil attack to attract more multicast blocks directly from the root. DCast builds proper incentives into the protocol so that each such deviation either is not rational or will not bring down the utilities of the non-deviators below the safety-net utility.

The incentives basically instantiate the ideas of pairwise entry fees and profitable interactions via the novel design of *debt-links* and *doins*. During the pull-based gossip, the propagation of a multicast block from one peer A to another peer B is always coupled with the propagation of a doin on an *unoccupied* debt-link from A to B. A debt-link from A to B is established by B sending $\mathcal{D}_{\text{link}}$ (e.g., 2.5) junk blocks to A. Fundamentally, this debt-link is a pairwise bandwidth entry fee paid by B. Notice that establishing the debt-link hurts the utility of both A and B. B may establish multiple debt-links from A, and there may simultaneously exist debt-links in the reverse direction from B to A. A debt-link is *unoccupied* when first established. After propagating a doin via that debt-link, the debt-link becomes *occupied* until the corresponding doin is *paid*.

A doin is a debt and can be *issued* by any peer. The current holder of a doin conceptually "owes" the issuer of the doin. Doins may circulate (i.e., be relayed) in the system and thus can be viewed as a special kind of bankless virtual currency. All doins issued within one interval *expire* at the beginning of the next interval, after which new doins will be issued. A peer holding an expired doin will pay for that doin by sending the doin issuer $\mathcal{D}_{\text{pay}}$ (e.g., 2) multicast blocks. Our earlier idea of profitable interactions is achieved by properly setting $\mathcal{D}_{\text{pay}}$: Under proper $\mathcal{D}_{\text{pay}}$, it will be profitable for a peer to issue/relay a doin, assuming that the doin is later properly paid.

We will present later the protocol level details on doin circulation, doin payment, and freeing up debt-links, as well as a formal theorem. For now, let us first obtain some intuition on the incentives in this design.

4.3 DCast Intuition

For better understanding, we for now assume i) infinite number of intervals in the multicast session, ii) no message losses, and iii) no control message overhead (i.e., a bit sent is a bit in either some multicast block or some junk block). Section 4.4 will explain how these assumptions can be removed.

We set the parameters in DCast such that $\max(1, \sigma) < \mathcal{D}_{\text{pay}} < \mathcal{D}_{\text{link}} < \mathcal{D}_{\text{root}}$. Recall from Section 3 that the constant σ is such that for most colluding peers, the benefit of receiving σ multicast bits on expectation exceeds the cost of sending/receiving one cost bit. We expect σ to be relative small (e.g., <2) in practice: Even in cases where the colluding peers may be able to obtain multicast blocks from each other with lower (but non-zero) bandwidth cost, σ will be above 2 only if their utility gain is above 100%, as compared to other peers.

Roughly speaking, the safety-net utility offered by DCast is such that i) a non-deviator will receive sufficient erasure-coded multicast blocks to decode all the video frames,[4] and ii) a non-deviator sends no more than $\mathcal{D}_{\text{root}}$ cost bits for each multicast bit received. Since peers in overlay multicast usually care much more about receiving the data than the overhead of sending data, we expect such utility to be sufficiently good as a safety-net utility for those non-deviators. Moreover, $\mathcal{D}_{\text{root}}$ cost bits is simply a worst-cast bound: In our experiments later, non-deviators can send as few as 1.077 cost bits for each multicast bit received. In comparison in previous designs [9, 14, 15, 18, 22] without the safety-net guarantee, a non-deviator would fail to obtain any multicast data at all in the presence of collusion, *regardless of how many cost bits it sends*.

Incentivizing doin payment. The first key incentive in DCast is using debt-link establishment cost as an effective punishment against peers who do not pay for doins. A peer B establishes debt-links from other peers purely for the purpose of pulling blocks from them. Since establishing a debt-link incurs overhead on B and since B has a choice over whether or not to establish a debt-link, a rational B will establish a debt-link only if the debt-link's existence is necessary to maintain its utility. If B does not pay for a doin, then the corresponding debt-link will remain occupied, and B may need to establish another debt-link to compensate. Doing so is less desirable than paying for the doin since $\mathcal{D}_{\text{pay}} < \mathcal{D}_{\text{link}}$.

Such argument holds for colluding peers as well. A colluding peer can obtain multicast blocks from other colluding peers. But if the colluding peer does establish an incoming debt-link from a non-deviator, it indicates that the peer is not able to rely on other colluding peers only, and has to pull blocks from some non-deviators via debt-links.

Incentivizing doin issuance/relay. Establishing a debt-link from A to B involves B sending junk blocks to A, and thus gives A disincentive to accept such debt-link establishment and to later issue/relay doins. DCast solves this problem by setting $\mathcal{D}_{\text{pay}} > \max(1, \sigma)$ and by properly re-using debt-links. Under such $\mathcal{D}_{\text{pay}}$, A makes some constant profit each time a doin is issued/relayed on a debt-link and then paid. Re-using the debt-link a sufficient number of times during the multicast session will then enable the accumulated profit to exceed the initial setup cost of the debt-link. This in turn incentivizes A to accept debt-link establishments and to send multicast blocks when non-deviators pull from them. In particular, such incentive also applies to deviators — while they might be able to disseminate data among themselves more efficiently than with non-deviators, under $\mathcal{D}_{\text{pay}} > \max(1, \sigma)$, issuing doins and then getting doin payments from non-deviators incur even smaller cost.

DCast does not provide any mechanism for peers to negotiate $\mathcal{D}_{\text{pay}}$, for two reasons. First, we do not believe that it is practical for users to negotiate the payment price. Second, setting a fixed price precludes the possibility of monopoly pricing. Namely, since the non-deviators will only make payments by the system-set price,

[4]Strictly speaking, even when all peers are non-deviators, random gossiping may still fail to achieve such a property, with some vanishingly small probability. It is however trivial to fully qualify our discussion by adding a "with high probability" condition.

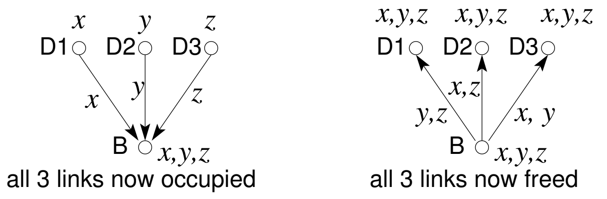

Figure 1: A non-deviator B pulls three blocks (x, y, and z), together with three doins, from three deviators D_1, D_2, and D_3. B then uses these blocks to pay off the exact three doins accompanying the three blocks under $\mathcal{D}_{\text{pay}} = 2$. The arrows in the figure are messages. The debt-links from D_1, D_2, and D_3 to B are not shown in the figure.

the deviators simply will not get a payment (and will hurt their own utilities) if they ask for a higher price.

Ability to pay debts. So far we have intuitively explained that a peer has the incentive to pay for a doin. For the payment to actually occur, the peer needs to have i) enough multicast blocks to offer to the doin issuer, and ii) enough bandwidth to send those blocks. It may not be immediately clear why peers will have enough blocks to pay off their (potentially high-interest) debts. If we view each block as a dollar and if $\mathcal{D}_{\text{pay}} = 2$, then every dollar must be repaid with two dollars and it seems that such an "economy" is impossible to sustain. Fortunately, our situation is fundamentally different from a real economy in that a peer B can send the same block to *multiple* peers and thus use that same "dollar" to offset multiple debts. This even makes it possible for B to purely rely on blocks pulled from deviators to pay off the debts to those deviators, despite that pulling those blocks will incur more debts. For example in Figure 1, B starts without any blocks, and then pulls 3 blocks from 3 deviators (D_1, D_2, and D_3), respectively. Afterwards, B can use these 3 blocks to fully pay off the debts to the 3 deviators.

The example also illustrates what will happen if B does not have enough (upload) bandwidth to pay off the debts. Conveniently, if B does not have bandwidth to pay, then B will not have bandwidth to establish new debt-links either. In fact, once B has available bandwidth, it will prefer paying off the old debts instead of establishing new debt-links. This avoids the undesirable situation where a peer without enough bandwidth just keeps borrowing new debts without paying off the old debts.

Rational sybil attacks. All our reasoning above applies to both non-sybil peers and sybil peers, except for the discussion on the peer list maintained by the root. Sybil attack enables one rational user to occupy a large fraction of the peer list, with two effects. First, the sybil peers will be selected more often for receiving blocks directly from the root. But since $\mathcal{D}_{\text{link}} < \mathcal{D}_{\text{root}}$, directly receiving the blocks from the root even exceeds the cost of pulling them from other peers. Thus the user has no incentive to create sybil identities to attract more blocks from the root. The second effect is that the sybil peers will more likely be in the view of other peers, and thus attract debt-link establishments. By our design, debt-link establishment by itself actually hurts a peer. Rather, the peer makes a profit only when it issues/relays doins and when the doins are paid. Thus a rational user only has the incentive to create as many sybil peers as they have enough bandwidth to issue/relay doins. As long as the sybil peers do this, they will not have any negative impact other than increasing the system size.

Small social cost. To sustain collaboration, peers in our DCast design sometimes need to send junk blocks. Our later experiments will show that, as a nice property of DCast, the relative social cost incurred by these junk blocks actually tends to zero over time in a large scale system. At a high level, this is because i) the total number of debt links established quickly stabilizes and thus debt link establishments largely only incur junk blocks at the beginning of the multicast session, and ii) the number of junk blocks sent to the root is dwarfed by the total amount of multicast blocks exchanged in the whole system.

4.4 Practical Issues

This section explains how to remove the three assumptions in the previous section, and also discusses the effects of malicious peers.

Finite number of intervals. Assuming infinite number of intervals enabled us to avoid the well-known *end-game effect*. For example, if the peers know that there are exactly 10 intervals, then no peer will issue doins in the 10th interval since they will not be paid back. In turn, no peer will have incentive to pay the doins issued in the 9th interval and free up the occupied debt-links. Backward induction will cause such argument to cascade back to all the intervals. This end-game effect is quite fundamental and applies to all previous proposals [9, 13, 14, 15, 18, 22] on incentivizing overlay multicast, as well as to all kinds of repeated games such as the iterated prisoner's dilemma.

On the other hand, this effect is widely considered [16] to be an artifact of modeling instead of a good prediction of how rational peers will behave. There are many well-known ways [16] to avoid this effect. For example, it suffices [9] just to assume that in each interval, the peers *expect* (which may or may not correspond to what actually happens) that with constant positive probability, there will be at least one more interval. Alternatively, one can invoke the ϵ-Nash concept and assume that the extra small utility obtained by not issuing doins in the very last interval does not give sufficient incentive for the peers to deviate [16]. Since these are largely orthogonal to DCast, to focus on DCast, our following discussion will simply continue considering infinite number of rounds.

Message losses and other unexpected events. Properly setting the parameters in DCast can easily take care of potential message losses. Let p be an upper bound on the probability of messages being lost. (Note that we do *not* require independent message losses.) Then the incentives in the previous section will continue to hold as long as $\max(1, \sigma) < (1-p)\mathcal{D}_{\text{pay}} < (1-p)^2 \mathcal{D}_{\text{link}} < (1-p)^3\mathcal{D}_{\text{root}}$, as explained below.

Let us first consider the incentive for doin payment. Previously without message losses, $\mathcal{D}_{\text{pay}} < \mathcal{D}_{\text{link}}$ provided incentive for a peer to pay a doin instead of establishing a new link. Now with message losses, it is possible that the payment is lost and thus the link remains occupied after payment. Furthermore, the doin issuer and the doin holder may now have an inconsistent view regarding whether the payment has been made. All these, however, will not disrupt the incentive as long as $\mathcal{D}_{\text{pay}} < (1-p)\mathcal{D}_{\text{link}}$. Since the payment successfully reaches the doin issuer with probability at least $(1-p)$, each payment frees up at least $(1-p)$ debt-links on expectation. If the peer instead chooses to establish $(1-p)$ new debt-links, it needs to send at least $(1-p)\mathcal{D}_{\text{link}}$ junk blocks. Note that the cost may be even higher if messages get lost during debt-link establishments. Since $\mathcal{D}_{\text{pay}} < (1-p)\mathcal{D}_{\text{link}}$, the peer will prefer paying for doins (while knowing that the payment may get lost) instead of establishing new debt-links.

Similar arguments apply to the incentives for issuing/relaying doins (provided $\max(1, \sigma) < (1-p)\mathcal{D}_{\text{pay}}$) and the disincentives for attracting blocks from the root (provided $\mathcal{D}_{\text{link}} < (1-p)\mathcal{D}_{\text{root}}$). Finally, similar arguments also apply to other unexpected events such as peer failures: As long as we set the parameters properly based on an upper bound on the probability of such events happening, peer failures will not disrupt our incentives.

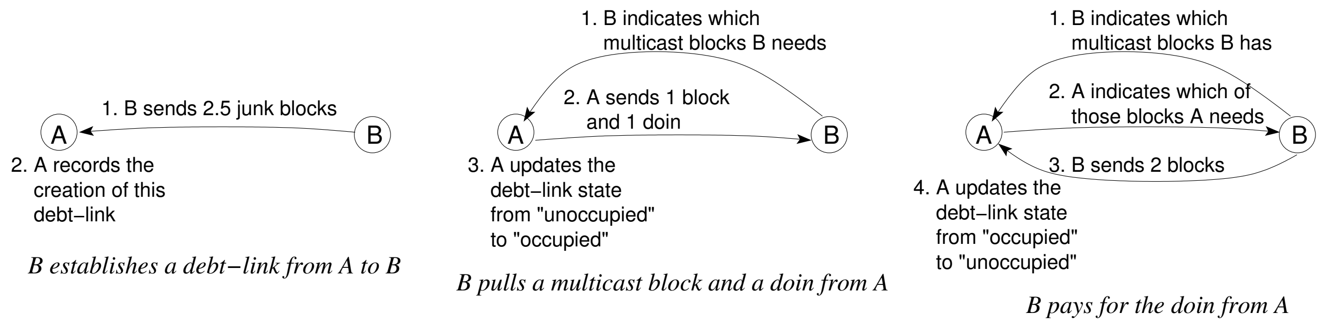
1. B sends 2.5 junk blocks
A
B
2. A records the creation of this debt−link
B establishes a debt−link from A to B
1. B indicates which multicast blocks B needs
2. A sends 1 block and 1 doin
A
B
3. A updates the debt−link state from "unoccupied" to "occupied"
B pulls a multicast block and a doin from A
1. B indicates which multicast blocks B has
2. A indicates which of those blocks A needs
A
B
3. B sends 2 blocks
4. A updates the debt−link state from "occupied" to "unoccupied"
B pays for the doin from A

Figure 2: Illustrating the key components of the basic one-hop protocol in Section 5.1. Here we use $\mathcal{D}_{\textbf{link}} = 2.5$ and $\mathcal{D}_{\textbf{pay}} = 2$, and only consider a single debt-link.

Control message overhead. Control message overhead is rather easy to accommodate: We again only need to set $\mathcal{D}_{\text{pay}}$, $\mathcal{D}_{\text{link}}$, and $\mathcal{D}_{\text{root}}$ properly so that the gaps among σ, $\mathcal{D}_{\text{pay}}$, $\mathcal{D}_{\text{link}}$, and $\mathcal{D}_{\text{root}}$ are still preserved after taking control overhead into account.

Effects of malicious peers. The safety-net guarantee does not protect the utility of a non-deviator if malicious peers bring down its utility (e.g., by sending it junk bits). Besides such direct DoS attacks on individual peers, overlay multicast by definition has a single root and thus is fundamentally vulnerable to DoS attacks on the root. Defending against these DoS attacks is beyond the scope of this work. However, we still aim to ensure that the attacker cannot significantly amplify its attack capacity (i.e., its attack bandwidth budget) by exploiting our DCast design, as compared to directly launching DoS attacks to the root or the peers. In particular, we intend to avoid grim trigger designs [13] where the whole system can "melt down" due to a single malicious peer sending a single message. The following discusses possible malicious attacks in DCast.

Malicious peers may attract multicast blocks from the root and then discard those blocks. For each such multicast block, the malicious peers need to send $\mathcal{D}_{\text{root}}$ junk blocks. If the malicious peers can send enough junk blocks to attract all the multicast blocks (notice that the root usually sends out as many multicast blocks as its bandwidth allows), then they can likely already directly DoS the root with such bandwidth.

Malicious peers may also interact with and attack the peers, in the following two ways. First, malicious peers may remain silent when other peers pull from them. To do so however, the malicious peers need to receive the junk blocks for debt-link establishment first, which makes the damage constrained by their attack bandwidth. Further because a peer monitors the multicast blocks it receives so far, the peer will simply establish new debt-links and pull from other peers when under such attack. Second, malicious peers may actively participate in doin issuance/propagation/payment. But the *worst* situation that can happen here is non-payment and for all the debt-links on the propagation chain to be permanently occupied. Under our full design with subintervals (Section 5.2), the total number of non-deviators that a doin can traverse is bounded (e.g., within 10). This in turn limits the damage that a malicious peer can cause in such a case. Note that in order to participate in doin issuance/propagation, the malicious peers will need to send/receive multicast blocks, which again makes the damage constrained by their attack bandwidth.

5. DCast PROTOCOL

This section elaborates the protocol level details on debt-link establishment, doin propagation, and doin payment. Algorithm 1 provides concise pseudo-code for these key procedures. For clarity, Section 5.1 first presents a basic version of the protocol, and then Section 5.2 describes the full DCast protocol. Also for clarity, the discussion in Section 5.1 and 5.2 will assume that the peers have no clock error — at the end of Section 5.2, we will explain how bounded clock errors can be trivially accounted for. Finally, we will describe the protocol as if there were no message losses. When messages are lost, no special action is needed and the intended receiver simply assumes that the sender did not send that message. As long as the protocol parameters are set properly as explained in Section 4.4, the incentives in DCast will not be disrupted by such message losses.

5.1 The Basic Protocol

Figure 2 illustrates the message exchanges in the key components of the basic protocol.

Message encryption and authentication. In DCast, every message is encrypted and authenticated via a MAC using a symmetric session key. In sharp contrast to typical virtual currency designs (e.g., [27]), DCast does not need any other crypto operations. Setting up these session keys is trivial since each peer has a public/private key pair. Specifically, a peer B may communicate with peer A when i) paying for doins issued by A, ii) establishing or freeing up debt-links from A, or iii) pulling multicast blocks from A. When paying for a doin, A's public key's hash is already contained in the doin's id (explained later). When establishing debt-links, B will contact some random IP addresses in its view to obtain those peers' public keys. Since here B just needs to obtain the public key of some random peers, it does not matter if the IP address is spoofed — that would just be equivalent to polluting B's view via a sybil attack, which we already discussed earlier. Finally, when freeing up debt-links or pulling blocks from A, B must already have A's public key during debt-link establishment.

Establishing debt-links. To establish a debt-link from A, B simply sends $\mathcal{D}_{\text{link}}$ junk blocks to A. No reply from A is needed. How many debt-links to establish from which peers is largely a performance issue, and DCast uses the following heuristic. During each of the first few rounds (e.g., 10), each peer B selects a distinct peer in its view and establishes a certain number (e.g., 20 to 80, depending on available bandwidth) of debt-links from that peer. Afterwards, B monitors the number of multicast blocks that it has received in recent rounds, and establishes additional debt-links if that number is low.

Pulling blocks and propagating doins. In each round, a peer B selects some random peer A that has debt-links to B, and pulls multicast blocks from A. To do so, B first sends to A a *summary* describing the multicast blocks that B already has. A then sends back those blocks that A has but B does not have, up to the number of unoccupied debt-links from A to B. For each block, A also passes to B the *id* of some doin. A doin's id includes the IP address

Algorithm 1 DCast routines for pulling blocks, sending blocks with doins, paying for doins, and freeing up debt-links. All messages are encrypted and authenticated (not shown). RLI means release local id, which serves to enable the two peers at the current hop to differentiate (potentially) concurrent Releases for the same debt-link when some peers deviate.

```
1:  /* Pulling multicast blocks */
2:  Send ⟨"data-request", summary⟩ to a random peer that has at
    least one unoccupied debt-link to me;
3:  Wait for ⟨"data-reply", (block, doin id, debt-link id) tuples⟩;

4:  /* Sending multicast blocks and doins */
5:  Upon receiving ⟨"data-request", summary⟩ from $B$:
6:      $S$ = set of blocks that I have and are not in summary;
7:      $k = \max(|S|$, number of unoccupied debt-links to $B$);
8:      if I hold less than $k$ doins, then issue new doins;
9:      Send back ⟨"data-reply", $k$ tuples of (block, doin id, debt-
    link id)⟩;

10: /* Paying for a doin */
11: for each expired doin that I hold do
12:     Send ⟨"pay-request", doin id, summary⟩ to doin issuer;
13:     Wait for ⟨"bill", doin id, blocks needed⟩;
14:     Send back ⟨"payment", doin id, $\mathcal{D}_{\text{pay}}$ blocks⟩;
15:     Wait for time $d$;
16:     Send ⟨"release", debt-link id, RLI⟩ to predecessor;
17: end for

18: /* Accepting a doin payment */
19: Upon receiving ⟨"pay-request", doin id, summary⟩:
20: if I can find in the summary at least $\mathcal{D}_{\text{pay}}$ blocks that I need
    then
21:     Send back ⟨"bill", doin id, blocks needed⟩;
22:     Wait for ⟨"payment", doin id, $\mathcal{D}_{\text{pay}}$ blocks⟩;
23:     Mark the doin as paid;
24: end if
25: Upon receiving ⟨"release", debt-link id, RLI⟩ and if I am the
    issuer of the corresponding doin:
26:     if the doin has been paid, then free up debt-link;
27:     else send back ⟨"denial", debt-link id, RLI⟩;

28: /* For a peer who previously relayed the doin */
29: Upon receiving ⟨"release", debt-link id, RLI⟩:
30:     Relay release to predecessor (with proper debt-link id and
        RLI);
31:     Set a timer of $2d \cdot$ (doin's r-stamp) for this release;
32:     if no denial within the timeout, then free up debt-link;
33: Upon receiving ⟨"denial", debt-link id, RLI⟩:
34: if the corresponding release has not timed out then
35:     Invalidate the release (i.e., will not free up debt-link);
36:     Relay denial to successor (with proper debt-link id and
        RLI);
37: end if
```

and the one-way hash of the public key of the doin issuer, a distinct *sequence number* assigned by the doin issuer, as well as the *issuing interval* indicating the interval during which the doin was issued. The total size of the doin id is less than 40 bytes (in comparison, multicast blocks are often 1KB or larger). The one-way hash function is publicly-known, and the hash enables the authentication of the doin issuer. One could directly include the public key itself, but a hash is shorter.

The above protocol with doins has several salient features as compared to virtual currency systems. First, our protocol is highly efficient and is the same as plain gossiping except for piggybacking the doin ids in A's reply. In particular, there is no need for A to obtain any signature from B to certify the acceptance of the debt. It does not matter if B later denies the debt — if B does not pay for the debt, A simply will not free up the corresponding debt-links. In fact, even if A obtained a signature from B, such signature would be of no use: There is no arbitrator in the system that can arbitrate the interaction and punish B. Second, the double-spending problem with virtual currency is completely avoided — a rational peer will prefer to issue a new debt instead of relaying the same debt to two different peers. Third, there is no need to protect the doin id from manipulation when a doin is relayed. Modifying any field in the doin id will be equivalent to "issuing" a new doin and not propagating the current doin.[5]

Paying for *one-hop* doins and freeing up debt-links. If a doin only circulated for one hop before being paid, the payment process would be simple. Namely, the current holder B of the doin will send to the doin issuer A the doin id and a summary of the multicast blocks that B has. If A can find $\mathcal{D}_{\text{pay}}$ multicast blocks that A needs, A tells B those blocks. (Otherwise B needs to try again later.) B then sends the blocks to A and completes the payment. After receiving the blocks, A frees up the debt-link.

Same as earlier, here B does not need to obtain a signature from A to certify the payment. A has no incentive not to free up the debt-link after the payment. After all, even if it does not free up the debt-link, B will not make a second payment.[6] Furthermore, not freeing up the debt-link would prevent A from making further profits on the debt-link.

5.2 The Full Protocol

Paying for *multi-hop* doins and freeing up debt-links. The situation becomes more complex when a doin traverses a chain of peers. The central difficulty here is the lack of incentive for the doin issuer to notify other peers on the chain to free up their corresponding outgoing debt-links. For example, imagine that a doin traverses three peers $A \rightarrow C \rightarrow B$ and then B pays A for the doin. C needs to be notified of the successful payment so that it can free up the debt-link to B. A has no incentive to notify C, since sending notification costs bandwidth. B does have the incentive to notify C, but B needs to present C a receipt from A to convince C. Unfortunately, A again has no incentive to send B such a receipt. The standard way of solving this problem is to use fair exchange protocols [10] so that A will only get the payment if it simultaneously gives B a receipt. Unfortunately, those heavy-weight multi-round protocols incur by far too much overhead for our purpose.

It turns out that by leveraging the rationality of the peers, a fair exchange is not necessary. Our key observation here is that while A has no incentive to confirm the payment, it does have the incentive to rebut a false payment claim. Namely, if B does not pay and falsely claims to C that B has paid, then A has the incentive to tell

[5]Of course, this also means that a deviator may "issue" doins on other peer's behalf. Our proof will show that a deviator has no incentive to do so, again because a doin is a debt.

[6]Formally, it is possible to model this interaction as a game with two Nash equilibria, and the completion of the payment publicly signals to the two peers the transition from one equilibrium to the other. Properly switching between two equilibria is a well-known technique [16] in mechanism design. Here we omit the tedious formalism. Also as shown earlier, unexpected message losses can be readily dealt with as well, as long as the loss probability is not excessive.

C that B has lied, because otherwise A will no longer get the payment. This observation leads to the following design where A does not send out a receipt — rather, it sends out a denial if necessary.

In DCast, after making the payment, B propagates a simple Release message backward along the chain to request the debt-links to be freed up. Each peer on the chain has clear incentive to relay the Release to its predecessor, since doing so is the only way to free up its incoming debt-link. When the Release reaches A, if A did not actually receive the payment, A will propagate a Denial message along the chain to prevent the debt-links from being freed up. A has clear incentive to do so, because otherwise A will no longer get a payment. Similarly, all the peers on the chain have the incentive to relay the Denial to make sure that the message reaches B (so that a rational B will make the proper payment), because otherwise their incoming debt-links will never be freed up.

Obviously, A may still propagate a Denial even though payment has been made. A closer examination shows that A actually has no incentive to do so. The reason is exactly the same as in our earlier scenario with one-hop doins: A will simply not obtain a second payment even if it propagates a Denial after a payment has been made. Finally in the above design, after receiving a Release, a peer needs to wait for a potential Denial within a certain timeout, before freeing up the debt-link. The following discusses how to properly set the timeout value.

Properly timing out when waiting for a potential Denial. Consider a chain of peers through which the doin has traversed. If every peer on the chain naively uses the same timeout, then a deviator may intentionally delay the Denial and cause the Denial to reach some peers but not others before the timeout. Conceptually to avoid this problem, peers farther away from the doin issuer need to have larger timeout. Hop counts would enable a peer to determine how far away it is from the doin issuer, but hop counts can be easily manipulated by the deviators. Instead, DCast uses the receiving time of a doin as a secure hop count that is guaranteed to be monotonic but not necessarily consecutive. Specifically, we divide each interval into a number (e.g., 10) of equal-length *subintervals*. A peer records the subinterval during which a doin is received as the doin's *r-stamp*. For a doin received in subinterval i, a non-deviator may only relay the doin in subinterval j when $j \geq i + 1$. In other words, a doin traverses at most one non-deviator in one subinterval.

A peer B with a doin r-stamp of r simply uses a timeout duration of $2dr$ after it forwards a Release to its predecessor C. Here d is a pessimistic upper bound on message propagation delay, and messages with delay larger than d will simply be treated as lost. As explained in Section 4.4, message losses will not disrupt the incentives in DCast as long as the loss probability is not excessive. If B and B's predecessor C are both non-deviators, then B's timer's duration will be at least $2d$ longer than C's. Thus if C receives the Denial in time, B is guaranteed to receive the Denial in time as well, if the message is not lost. The following simple lemma, whose proof is trivial and thus omitted, formalizes the properties of this design when no messages are lost:

LEMMA 1. Consider any consecutive sequence of non-deviators $A_1A_2...A_k$ that a doin traverses. With our above design, if A_1 receives (or generates, when A_1 is the doin issuer) a Denial before its local timer expires, then the Denial will reach every peer A_i ($1 \leq i \leq k$) before A_i's timer expires.

With deviators on the propagation chain, the chain will be divided into multiple consecutive sequences of non-deviators. Reasoning about each such sequence individually (as in Lemma 1) will be sufficient for our formal proof later.

Simultaneous exchange. The full DCast protocol further has the following optional but useful optimization. Imagine two peers A and B, each with a multicast block needed by the other party. These two blocks can be propagated using two doins, which will occupy a debt-link from A to B and a debt-link from B to A. As an optimization, DCast permits A and B to exchange these two blocks without eventually occupying any debt-links, if they both would like to do so. Specifically, when B pulls from A, A will propagate the block together with a doin as usual. If A is interested in performing simultaneous exchange, A sets a flag in the message and indicates which multicast block that A wants (based on B's summary). The flag tells B that if B returns the requested block by the next round, A will consider the doin as never having been propagated to B, and will free the debt-link from A to B immediately. If B does not return that block, this interaction will just be considered as a normal pull, and B will hold the doin.

Dealing with clock errors. Finally, our discussion in Section 5 so far has been assuming zero clock error. Non-zero clock errors will introduce two issues. First, when a peer A propagates a doin (issued during interval i) to another peer B, A may consider the current time to be in interval i, while B's current time is in interval $i + 1$ or $i - 1$. This problem is trivial to address — A will simply avoid propagating doins issued during interval i (and start issuing doins for interval $i + 1$) when it is rather close to the end of interval i. A peer should accept doins whose issuing interval is either the current interval or some future interval. A second and similar issue arises for subintervals: When A relays a doin with an r-stamp of x in the $(x + 1)$th subinterval (according to A's local clock) to B, B may believe that the current time is still in the xth subinterval. This is easily addressed by having A avoid relaying doins with an r-stamp of x during the first few seconds of subinterval $x+1$. Finally, clock errors will not affect our timeout design for waiting for a potential Denial, since there we rely on timer duration instead of clock readings.

6. FORMALLY ACHIEVING SAFETY-NET GUARANTEE

This section presents the main theorem on DCast's safety-net guarantee. The formalization is non-trivial, and we start from the notion of a *reference execution*.

Reference execution. We want to avoid defining any specific utility functions, since it is hard to predict the exact form of the peers' utility functions. Without a concrete utility function, however, we will not be able to assign a numerical value to the safety-net utility. Rather, we will use a *reference execution* as the reference point, where all peers follow a certain simple multicast protocol. Because there are no deviating peers in this reference execution, the utility achieved by each peer here is easy to understand, and one can readily plug in various utility functions to obtain instantiated utility values. We will then show that DCast's safety-net utility for a specific peer is the same as that peer's utility in the reference execution.

For our formal arguments next, we will assume a static setting where no peers join or leave on the fly. Our reference execution is the execution of a simple multicast protocol using pull-based gossiping. There are m users in the system, and user i has $x_i \geq 1$ identities (peers). The multicast proceeds in rounds, where in each round the root sends erasure-coded multicast blocks to some set of peers chosen arbitrarily. Also in each round, a peer selects a uniformly random peer out of the $\sum_{i=1}^{m} x_i$ peers from whom to pull multicast blocks. All messages are reliable. Because all peers are cooperative, there are no concepts such as debt-links or doins. The execution has a single parameter Ψ: For each multicast bit received, a peer sends Ψ cost bits to some special virtual sink peer.

The main theorem. Now we can present our main theorem. For clarity, here we will simplify away the issues discussed in Section 4.4, since Section 4.4 already explained how they can be properly taken into account. Specifically, for our main theorem, we will assume infinite number of rounds, no message losses, no peer failures, no control message overhead, and no malicious peers. With infinite number of rounds, we further assume that if the two peers incidental to a debt-link are willing to use the debt-link whenever opportunities arise, then that debt-link will be reused infinite number of times during the multicast session. This assumption is mainly for simplicity and is not necessary for the theorem to hold (see Section 7 for more discussion).

THEOREM 2. *Assume that:*

- *When pulling multicast blocks from another peer, a non-deviator establishes enough new debt-links, if needed, to pull all the blocks that it needs from that peer.*
- *During doin payment, the payer always eventually finds enough blocks that the payee does not have, so that the payment can be completed.*
- *At any point of time, a deviator gets at least those multicast blocks that it would get if it did not deviate.*

If we set the parameters in DCast such that:

$$\max(1, \sigma) < \mathcal{D}_{pay} < \mathcal{D}_{link} < \mathcal{D}_{root} \qquad (1)$$

where σ is defined as in Section 3, then DCast provides a safety-net guarantee where despite the rational deviation of any set of peers, a non-deviating peer will always obtain a utility no smaller than its utility in the reference execution with the same set of peers and with a parameter $\Psi = \mathcal{D}_{root}$. In other words, this utility in the reference execution is the safety-net utility.

To better understand the assumptions in the theorem, let us examine these assumptions one by one. First, a non-deviator clearly needs to establish enough debt-links in order to achieve a reasonable utility. Second for doin payment, in our later experiments, we observe that over 99.95% of the doins can be properly paid. The last assumption is needed because if the deviators receive fewer multicast blocks after deviating, then the non-deviators will not be able to pull enough multicast blocks even if the deviators entirely cooperate with the pull.

Proving Theorem 2. We leave the full proof to the Appendix, and briefly explain here how it is obtained. As discussed in Section 3, we need to consider only pareto-optimal collusion strategies. At a high level, our proof first shows that a pareto-optimal collusion strategy must be *non-damaging*. Roughly speaking, under a *non-damaging* collusion strategy, for every doin that involves at least one deviator, the non-deviators' utilities resulted from that doin's issuance/propagation/payment are the same as or better than when the involved deviators exactly followed the doin issuance/ propagation/payment protocol. Intuitively, a pareto-optimal collusion strategy in DCast must be non-damaging because if a deviator does some "damaging" action to the non-deviators, it will hurt its own utility as well.

Next based on Equation 1 in the theorem, we show that issuing doins is profitable under proper debt-link reuse, and thus the deviators will be willing to issue doins and propagate multicast blocks to the non-deviators. Furthermore because the collusion strategy is non-damaging, the utility of the non-deviators during doin issuance/propagation/payment will be properly protected. This enables us to reason about the total cost bits that a non-deviator needs to send/receive for obtaining the multicast blocks. Combining all the above will complete the proof (see Appendix).

7. IMPLEMENTATION AND EVALUATION

The main purpose of our implementation and experiments is to supplement our formal proof in Section 6. Specifically, our experiments aim to i) quantify/illustrate the end guarantees of DCast, ii) quantify the overheads in DCast, and iii) validate some of the simplifying assumptions made in Theorem 2. Note that our experiments do not serve to show DCast's improvement over previous designs, since the improvement (i.e., robustness against rational collusions and sybil/whitewashing attacks) is qualitative and has been proved in Theorem 2.

To this end, we have implemented both a prototype and a simulator for DCast. Our prototype serves to validate the feasibility of DCast in practice, and also to validate the accuracy of our simulator. The simulator on the other hand, enables us to perform experiments of larger scale. The DCast prototype is implemented in Java 1.6.0 using regular TCP for communication. To strike a balance between performance and ease of implementation, we use an event-driven architecture with non-blocking I/O in bottleneck components, and a simpler multi-threading architecture elsewhere.

Unless otherwise mentioned, all our experiments use the following setting. The multicast session uses 10,000 peers, and is one hour long with a streaming rate of 200Kbps. The session has 60 one-minute intervals where each interval has 30 two-second rounds. The size of each multicast block is 1KB. We set $\mathcal{D}_{\text{link}} = \mathcal{D}_{\text{pay}} + 0.5$ and $\mathcal{D}_{\text{root}} = \mathcal{D}_{\text{pay}} + 1$, with $\mathcal{D}_{\text{pay}}$ ranging from 2 to 4. We do not expect that $\mathcal{D}_{\text{pay}} > 2$ is needed in practical scenarios, and thus those settings mainly serve as stress tests. In each round, the root sends 100 erasure-coded multicast blocks to 100 randomly chosen peers.[7] Any 50 out of these 100 blocks are sufficient to decode the video frames. The deadline of the multicast blocks is 20 rounds after they are sent from the root.

In the next, we first elaborate our large-scale simulation results, and then present the validation results of the simulator using our prototype. The end-to-end metric for video streaming is usually delivery_rate, which is the fraction of rounds that an average peer receives the corresponding multicast blocks by the deadline and thus can render the video frames for those rounds. In *all* our simulation experiments (even with colluding deviators), we observe a delivery rate for the non-deviators of at least 99.95% and thus we will not discuss delivery_rate further. The following presents results that help us to gain further insights into the social cost and the safety-net guarantee of DCast.

No deviators: Social cost. We first investigate the social cost (i.e., where junk blocks are involved) of DCast when no peer deviates. This social cost has two components: debt-link establishment costs and the cost incurred by sending junk blocks to the root. The first component by far dominates the second one. Figure 3 plots the average number of debt-links established by each peer, denoted as links_established. The figure shows that for the first few rounds, there is a sharp jump since all peers are establishing new

[7]We intentionally keep the load on the root light. This not only reserves enough bandwidth for the root to receive junk blocks, but also stress tests our design. Under the given streaming rate, the total load on our root (including the load incurred by the junk blocks) is less than half of the load in similar prior experiments with gossip-based multicast [14].

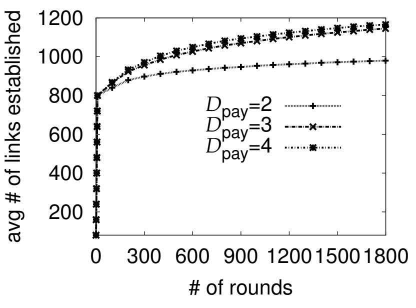

Figure 3: Average number of debt-links created by a peer.

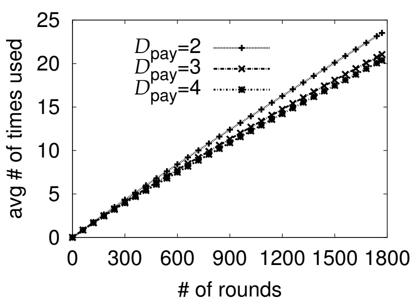

Figure 4: Average number of times that a debt-link is used.

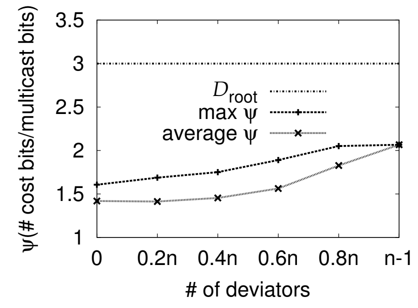

Figure 5: Example of safety-net guarantee.

debt-links. Then the curve grows rather slowly, and remains below 1,200 in the end.[8] On average each peer establishes 0.67 debt-links in each round. Such overhead is rather small and involves only $0.67\times\mathcal{D}_{\text{link}}$ junk blocks, as compared to 50 multicast blocks sent/received by an average peer in each round. Thus the relative overhead incurred by the junk blocks is roughly $0.67\times\mathcal{D}_{\text{link}}/50$, or between 3.35% and 6.03% under our experimental parameters.

No deviators: Number of times a debt-link is used. Next we examine the number of times that a debt-link is used[9] (denoted as times_used) and payment success rate (defined below) in DCast. These two quantities capture two key assumptions behind Theorem 2. The proof for Theorem 2 assumes that a debt-link is used infinite number of times, as long as the two peers incidental to the debt-link are willing to use it whenever opportunities arise. However, since the total number of rounds is not actually infinite, the usage time will not be infinite either. If times_used is overly small, the accumulated profit of issuing/relaying doins might not be able to offset the debt-link establishment cost. Figure 4 shows that an average debt-link is used over 20 times during the one-hour multicast session. This value is far larger than what is needed to offset the initial debt-link establishment cost: Establishing a debt-link from A to B involves $\mathcal{D}_{\text{pay}}+0.5$ junk blocks, while every time the debt-link is used, A sends one multicast block and in return either receives $\mathcal{D}_{\text{pay}}$ multicast blocks (if A issues a doin) or eliminates $\mathcal{D}_{\text{pay}}$ blocks of debt (if A relays a doin). Thus, times_used only needs to satisfy the following inequality in order for Theorem 2's proof (see Appendix) to hold: $(\text{times\_used} + \mathcal{D}_{\text{link}}) \times \max(1,\sigma) < \text{times\_used} \times \mathcal{D}_{\text{pay}}$. As a numerical example, for $\sigma = 1$ and $\mathcal{D}_{\text{pay}} = 2$, times_used only needs to reach 2.5 to offset the debt-link establishment cost.

No deviators: Payment success rate. *Payment success rate* (denoted as pay_succ_rate) is the fraction of expired doins that are paid by the time that the multicast session ends. While a rational peer always has the incentive to pay, pay_succ_rate may still be below 100% if it has no multicast blocks to offer to the doin issuer. If pay_succ_rate is so low such that $\text{pay\_succ\_rate} \times \mathcal{D}_{\text{pay}} < \max(1,\sigma)$, then the peers may no longer have incentive to issue doins. We observe a pay_succ_rate of above 99.95% in all our experiments, which is clearly large enough as long as we include a small extra gap between $\mathcal{D}_{\text{pay}}$ and $\max(1,\sigma)$.

Effect of deviators. So far our results on links_established, times_used, and pay_succ_rate are only for scenarios where no peer deviates. For times_used and pay_succ_rate, notice that larger times_used and pay_succ_rate benefit *all* peers. Thus the deviators actually have the incentive to further increase these two quantities, if possible. So at the very least, we do not expect that the deviators will behave in such a way to significantly decrease them, as compared to what we have observed in our earlier experiments. On the other hand with deviators, links_established and the corresponding social cost may increase, as in our later experiments on the safety-net guarantee. In those experiments, the debt link establishment cost will be directly captured in (i.e., deducted from) the utility achieved by the non-deviators.

Having 0 through $n-1$ deviators: Safety-net utility under an example collusion strategy. We next aim to illustrate the safety-net guarantee offered by DCast under an example collusion strategy, with $\mathcal{D}_{\text{pay}} = 2$. We do not intend to be exhaustive here — experimental methods by definition cannot cover all collusion strategies. It is not meaningful to consider "representative" collusion strategies either, because the human attacker is intelligent and may devise novel collusion strategies that we do not expect today. Ultimately, the safety-net guarantee has to be proved, as we did in Theorem 2. Our experiments here purely serve as an example.

In this example collusion strategy, a deviator never pulls data from a non-deviator, to avoid establishing debt-links, receiving doins, and paying for doins. The deviators collude by pulling data from each other, without the constraints of debt-links and doins. Doing so enables them to disseminate the multicast blocks faster and with lower overheads. Since issuing doins is profitable, a deviator will still issue doins to non-deviators and still accept doin payments.

Our experiments vary the number of deviators from 0 to $n-1$, where n is the total number of peers. For each non-deviator, we record the ratio between the total number of cost bits sent/received and the total number of multicast bits received — this ratio is essentially the Ψ parameter in the safety-net utility. Note that since the delivery_rate we observe is close to 100%, the non-deviators almost always receive enough multicast blocks to decode the video frames in each round. Figure 5 plots both the average and the maximum Ψ across the non-deviators, which is always below $\mathcal{D}_{\text{root}}$ and thus consistent with Theorem 2. Even with the extreme case of $n-1$ deviators, by running DCast, the single non-deviator still enjoys a delivery_rate of above 99.95%, while paying about 2 cost bits for each multicast bit received. Recall that in multicast, it is usually far more important to obtain the data (e.g., to watch the movie) than reducing bandwidth consumption (i.e., to incur fewer cost bits).

As expected, Ψ increases with the number of deviators in Figure 5. Such increase is the combined result of paying for more doins, issuing fewer doins, and establishing more debt-links. But even when there is no deviator, Ψ is still around 1.5. Here debt-link establishment and sending junk blocks to the root only contribute

[8]With the simultaneous exchange optimization, the number of debt-links needed by a peer can be smaller than the number of multicast blocks received during an interval (which is $50 \times 30 = 1500$).

[9]We exclude debt-link utilization in simultaneous exchanges, which only makes our results more pessimistic.

metric	prototype	simulation
delivery_rate	99.93%	99.996%
links_established	1114	890
times_used	13.5	12.7
pay_succ_rate	99.97%	1.0

Table 1: Validating the simulator using the prototype.

about 0.056 and 0.0012 to Ψ, respectively. The remaining part is for sending multicast blocks to other peers. This part is larger than 1.0 because with simultaneous exchange, sometime multiple peers may send redundant blocks to the same target peer. Our simulator is particularly pessimistic in this aspect — the multicast blocks received by the target peer will only be visible in the next round (i.e., 2 seconds later). This causes a relatively large number of redundant blocks. Further simulation confirms that if the multicast blocks received are immediately visible, then Ψ will drop from 1.5 to 1.077 as expected.

Validating the simulator using the prototype. To validate the accuracy of our simulator, we run our DCast prototype on the Emulab testbed, with 180 peers and $\mathcal{D}_{\mathsf{pay}} = 2$. While our protocol is an *overlay* multicast protocol, it is infeasible for us to emulate a network topology, which would require us to emulate all the routers between each pair of the peers. Instead, we artificially add realistic wide-area communication delays for the messages, by mapping each peer to a random node in the King Internet latency dataset [7] and using the latency there as the delay values.

Table 1 compares the results from a half-hour multicast session using our prototype versus the results from our simulator under the same setting. The value of times_used is rather similar in the two cases. While delivery_rate and pay_succ_rate on Emulab are slight lower than in the simulation, their absolute values remain rather high. The difference is mainly caused by unexpected delays on some Emulab nodes, which our simulator is unable to capture. Finally, the Emulab experiment establishes about 25% more debt-links than the simulation experiment. This is due to the processing delay on some Emulab nodes, which causes peers to establish more debt-links to ensure that they can get the multicast blocks in time. Since debt-link establishment contributes to a rather small component of Ψ (e.g., 0.056 out of 1.077), we believe that the safety-net utility observed in the simulation is still quite accurate.

8. CONCLUSION

The state of the art in overlay multicast is rather weak when it comes to security. This paper aims to address one of the key vulnerabilities in all prior overlay multicast protocols — the vulnerability against the collusion and sybil/whitewashing attacks by rational users. To this end, we first introduced the novel notion of a *safety-net guarantee* in a game theoretic context, which focuses on protecting the utility of the non-deviators. We then presented the *DCast* multicast protocol that uses a novel mechanism with debts circulating on pre-established debt-links. This mechanism enabled us to overcome two fundamental challenges introduced by rational collusion. We formally proved that the protocol offers a safety-net guarantee, and further demonstrated via prototyping and simulation the feasibility and safety-net guarantee of our design in practice.

9. ACKNOWLEDGMENTS

We thank Binbin Chen and the CCS anonymous reviewers for helpful feedbacks. This work is partly supported by National University of Singapore FRC grant R-252-000-406-112, and partly supported by the Intel Science and Technology Center for Cloud Computing (ISTC-CC).

10. REFERENCES

[1] E. Adar and B. Huberman. Free riding on Gnutella. *First Monday*, 5(10), 2000.
[2] C. Aperjis, M. Freedman, and R. Johari. Peer-assisted content distribution with prices. In *CoNext*, 2008.
[3] L. P. Cox and B. D. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *SOSP*, 2003.
[4] J. Douceur. The Sybil attack. In *IPTPS*, 2002.
[5] J. Feigenbaum and S. Shenker. Distributed algorithmic mechanism design: Recent results and future directions. In *DIALM*, 2002.
[6] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. In *SOSP*, 2003.
[7] K. Gummadi, S. Saroiu, and S. Gribble. King: Estimating latency between arbitrary internet end hosts. In *SIGCOMM*, 2002.
[8] A. Hayrapetyan, E. Tardos, and T. Wexler. The effect of collusion in congestion games. In *STOC*, 2006.
[9] I. Keidar, R. Melamed, and A. Orda. EquiCast: Scalable multicast with selfish users. In *PODC*, 2006.
[10] S. Kremer, O. Markowitch, and J. Zhou. An Intensive Survey of Fair Non-Repudiation Protocols. *Computer Comm.*, 25(17), 2002.
[11] R. Landa, D. Griffin, R. Clegg, E. Mykoniati, and M. Rio. A sybilproof indirect reciprocity mechanism for peer-to-peer networks. In *INFOCOM*, 2009.
[12] D. Levin, K. LaCurts, N. Spring, and B. Bhattacharjee. BitTorrent is an Auction: Analyzing and Improving BitTorrent's Incentives. In *SIGCOMM*, 2008.
[13] D. Levin, R. Sherwood, and B. Bhattacharjee. Fair File Swarming with FOX. In *IPTPS*, 2006.
[14] H. C. Li, A. Clement, M. Marchetti, M. Kapritsos, L. Robison, L. Alvisi, and M. Dahlin. Flightpath: Obedience vs. choice in cooperative services. In *OSDI*, 2008.
[15] H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. BAR Gossip. In *OSDI*, 2006.
[16] G. J. Mailath and L. Samuelson. *Repeated Games and Reputations*. Oxford University Press, 2006.
[17] A. Nandi, T.-W. J. Ngan, A. Singh, P. Druschel, and D. S. Wallach. Scrivener: Providing incentives in cooperative content distribution systems. In *Middleware*, 2005.
[18] T.-W. J. Ngan, D. S. Wallach, and P. Druschel. Incentives-compatible peer-to-peer multicast. In *P2P Econ*, 2004.
[19] N. Nisan, T. Roughgarden, E. Tardos, and V. V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, 2007.
[20] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in BitTorrent. In *NSDI*, 2007.
[21] M. Piatek, T. Isdal, A. Krishnamurthy, and T. Anderson. One hop reputations for peer to peer file sharing workloads. In *NSDI*, 2008.
[22] M. Piatek, A. Krishnamurthy, A. Venkataramani, R. Yang, and D. Zhang. Contracts: Practical Contribution Incentives for P2P Live Streaming. In *NSDI*, 2010.
[23] M. Reiter, V. Sekar, C. Spensky, and Z. Zhang. Making peer-assisted content distribution robust to collusion using bandwidth puzzles. In *ICISS*, 2009.
[24] S. Saroiu, P. Gummadi, and S. Gribble. Measuring and Analyzing the Characteristics of Napster and Gnutella Hosts. *Multimedia Systems Journal*, 9(2), 2003.
[25] N. Tran, J. Li, and L. Subramanian. Collusion-resilient Credit-based Reputations for Peer-to-peer Content Distribution. In *NetEcon*, 2010.
[26] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. Karma: A secure economic framework for peer-to-peer resource sharing. In *P2P Econ*, 2003.
[27] B. Yang and H. Garcia-Molina. PPay: Micropayments for peer-to-peer systems. In *CCS*, 2003.
[28] H. Yu. Sybil defenses via social networks: A tutorial and survey. *ACM SIGACT News*, 42(3), 2011.
[29] H. Yu, P. B. Gibbons, and C. Shi. Brief Announcement: Sustaining

Collaboration in Multicast despite Rational Collusion. In *PODC*, 2011.

[30] Z. Zhang, S. Chen, and M. Yoon. MARCH: A Distributed Incentive Scheme for Peer-to-Peer Networks. In *INFOCOM*, 2007.

APPENDIX

This appendix proves Theorem 2 from Section 6. First, we want to prove that a pareto-optimal collusion strategy must be *non-damaging*, as defined in the following:

DEFINITION 3. *A collusion strategy is called* non-damaging *if for every non-deviator (all steps below refer to steps in Algorithm 1 in Section 5):*

1. *If the non-deviator issues a new doin, then for that doin it successfully executes Step 19.*
2. *If it executes Step 19 for a doin (which may or may not correspond to a doin that it has issued), then for that doin it successfully executes Step 20-23.*
3. *If it issues a new doin, then for that doin it successfully executes Step 25-26.*
4. *It never executes Step 27.*
5. *If it holds a doin that has expired, then for that doin it successfully executes Step 12-16 exactly once, and the corresponding incoming debt-link will be freed.*
6. *If it relays a doin, then for that doin it successfully executes Step 29-32, and the corresponding incoming debt-link will be freed.*
7. *It never executes Step 33-37.*
8. *It never receives non-protocol messages (i.e., messages not specified by the DCast protocol).*

THEOREM 4. *Under the setting and the assumptions for Theorem 2, a pareto-optimal collusion strategy must be non-damaging.*

Proof: We need to introduce some formal notions. An *N-sequence* is a sequence of non-deviators that a doin traverses. An N-sequence is *maximal* if it is not part of another N-sequence. A *segment* is an N-sequence prepended by the deviator (if any) that sends the doin to the first peer in the N-sequence, and appended by the deviator (if any) that receives the doin from the last peer in the N-sequence. The first peer (which can be either a deviator or a non-deviator) of a segment is called the *head* of the segment, and the last peer is called the *tail*. By definition of a segment, all non-deviators on a segment see the same doin id, and that doin id is called the doin id of the given segment. Now consider any given pareto-optimal collusion strategy α, and we will show that it satisfies all the properties needed to be non-damaging.

Property 8. Non-protocol messages will always be ignored by a non-deviator, and thus their only effect is to reduce the utility of both the sender and the receiver of the messages. We claim that in α no deviators will send non-protocol messages to non-deviators, because otherwise α will be dominated by some other collusion strategy (which avoids sending these messages).

Property 1. Consider any non-deviator A that issues a doin, and the corresponding segment starting from A with that doin id. (Since a deviator might issue doins on other peers' behalf, we have not yet proved that there is exactly one segment with that doin id. However, even if there were multiple segments with the given doin id, exactly one of those will start from A.)

If the tail of the segment is a non-deviator, then obviously it will initiate a payment and cause A to execute Step 19. If the tail is a deviator D, let D's predecessor on the segment be B (B can be A itself), which must be a non-deviator by the definition of a segment. Assume by contradiction that A never executes Step 19. Then A will never mark the doin as "paid". In turn, the debt-link from B to D will never be freed. The reason is that to free the debt-link, B must receive a Release message, and then must not receive a Denial message within the timeout. But because the Release message will reach A, and A will generate a Denial message immediately, Lemma 1 tells us that B will receive the Denial message before timing out and thus will not free the debt-link from B to D. Since the debt-link will not be freed, D will need to establish another new debt-link. On the other hand, if we consider a second collusion strategy β where D simply makes the payment properly to A, D's utility will be better under β than under α, since $\mathcal{D}_{\text{pay}} < \mathcal{D}_{\text{link}}$. This contradicts with the fact the α is pareto-optimal.

Property 2. If a non-deviator A executes Step 19 for some doin, then we claim that the sender B (regardless of whether it is a non-deviator or deviator) of the "pay-request" message must enable A to complete Step 20-23. The reason is that the only effect of Step 19 is to trigger A to send the message at Step 21, which is of no use to B unless Step 23 is completed. If B does not enable A to complete Step 22 (implying that B is a deviator), then not sending the "pay-request" message would improve B's utility, rendering α non-pareto-optimal.

Property 3. Similar to the proof for Property 1.

Property 4. If a non-deviator A executes Step 27, then the corresponding segment must end with some deviator D. This Denial message must have been triggered by some previous Release message from D, since all other peers on the segment are non-deviators. Notice that the only effect of a Release message is to free debt-links. But because A executes Step 27, by Lemma 1 none of the debt-links will be freed. We thus claim that D would never send the original Release message in the first place, since otherwise not sending the Release message would improve D's utility and would make α non-pareto-optimal.

Property 5. Consider any non-deviator A that holds an expired doin. Notice that the doin issuer (as shown in the doin's id) may be different from the head of the corresponding segment. If the doin issuer is a non-deviator, then A can always complete the payment and then propagate a Release message to its predecessor, causing the debt-link to be freed. In particular, even if the head of the segment is a deviator D, D would not send a Denial message. The reason is that the only effect of sending a Denial message is to cause some debt-links to be permanently occupied, as well as incurring some overhead to every peer on the segment (including D itself). Since A will never make a second payment, the only effect of causing the debt-links to be occupied is to force the non-deviators to establish new debt-links and incur some extra debt-link establishment overhead. This means that D's sending the Denial message will simply decrease the utility of some peers, including itself. Since α is pareto-optimal, D will not do so.

If the doin issuer is a deviator D, we claim under α, D will always accept the payment because it always improves D's utility if it does so. For the same reason as above, no Denial message will be sent and A's incoming debt-link will be freed. Finally, it is possible for the doin issuer to be *non-existent*, when the head of the segment is a deviator D. A doin has a *non-existent* issuer if either the IP address in the doin's id does not correspond to any peer, or the peer at that IP address does not have the corresponding private key for the public key in the doin's id. Obviously, a doin with a non-existent issuer cannot be paid. We claim that because the collusion strategy α is pareto-optimal, the doin issuer will never be non-existent. The reason is that if it were non-existent, then D should just replace the doin issuer with itself, and accept payment later. This must improve D's utility, making α non-pareto-optimal.

Properties 6 and 7. Consider any non-deviator A that relays a doin, and consider the corresponding segment. If the tail of the segment is a non-deviator, then the tail will attempt to pay. By Property 5, the tail will successfully make the payment and propagate a Release message backward. We have also shown above (in the proof for Property 5) that there will not be a Denial message. Next if the head of the segment is a non-deviator, then we have shown earlier (for Properties 1, 2, 3, and 4) that a Release message will be propagated to the head (through A), and there will be no Denial message.

The only remaining case is when the head and the tail are two deviators D_1 and D_2, respectively. We first prove that there will not be a Denial message from D_1, by enumerating two possibilities:

- If D_2 did not previously propagate a Release message, then obviously D_1 would never send a Denial message. The reason is that such message will simply be ignored, and its only effect is to bring down the utility of D_1 and D_1's successor on the segment. Since α is pareto-optimal, D_1 will not do so.
- If D_2 did previously propagate a Release message on the segment, assume by contradiction that D_1 sends a Denial message to its successor B before B's timer expires. Because all peers on the segment, except D_1 and D_2, are non-deviators, Lemma 1 tells us that all these peers will receive the Denial message before their timers expire. Thus none of the debt-links will be freed, and the only effect of this Denial message is to cancel out the earlier Release message. We can now construct a second collusion strategy β by modifying α so that D_2 does not send the initial Release message. Doing so clearly reduces the cost bits sent/received by D_1 and D_2, which would imply that α is not pareto-optimal.

Next we prove that there is a Release message from D_2 to its predecessor C on the segment, via a contradiction. If there is no Release message, it means that D_2's incoming debt-link cannot be freed, and D_2 needs to establish a new debt-link. Consider a second collusion strategy β where D_2 makes a payment to D_1 and then propagates a Release. Compared to α, β improves the utility of both D_1 and D_2, which would make α non-pareto-optimal. □

Now we are ready to prove Theorem 2.

Proof for Theorem 2: By the definition of safety-net guarantee, we only need to consider pareto-optimal collusion strategies. Consider any given pareto-optimal collusion strategy α. We need to show that a non-deviator will achieve at least as good utility under α in the DCast execution as it would in the reference execution. We set the reference execution such that in each round, the root sends multicast blocks to exactly the same set of peers as in the DCast execution. We can do this because the reference execution allows this set of peers to be arbitrarily chosen.

We will first prove that at the end of any round, each peer (either non-deviator or deviator) in the DCast execution gets at least those multicast blocks that it gets in the reference execution. We prove via an induction on the number of deviators in the DCast execution. The induction base for zero deviator obviously holds. Now assume that the statement holds when the number of deviators is k. We consider the scenario with $k+1$ deviators, and prove the statement by contradiction. Let r be the first round at the end of which some peer does not get all those blocks that it gets in the reference execution. This peer is either a deviator or a non-deviator:

- If it is a deviator D, by the last assumption in Theorem 2, D must get at least those multicast blocks that D would get if D did not deviate. Thus let us consider a second DCast execution where D does not deviate and is a non-deviator. This second execution has only k deviators. By inductive hypothesis, every peer (including D) in this second DCast execution gets at least those multicast blocks that it gets in the reference execution. Thus in the original DCast execution with $k+1$ deviators where D is a deviator, D must also get at least those multicast blocks.
- If it is a non-deviator A, we will prove that A must have pulled from a deviator in round r, via a simple contradiction: If A pulled from another non-deviator B in round r, by the definition of r, at the end of round $r-1$, B has all those blocks that B has in the reference execution. Thus during round r, B must be able to propagate to A all those blocks that B propagates to A in the reference execution. This would contradict with the fact that at the end of round r, A does not get all the blocks that A gets in the reference execution.

  Thus A must have pulled from a deviator D in round r. Again by the definition of r, at the end of round $r-1$, D has all those blocks that D has in the reference execution. A always establishes enough debt-links to pull from D. Next we show that D will always choose to propagate as many multicast blocks as possible, via those debt-links. Consider a debt-link from D to A, and let x denote the total number of times that this debt-link will be used during the multicast session, if D issues or relays doins using the debt-link whenever possible. The total cost to D, related to this debt-link, will be the same as sending $x + \mathcal{D}_{\text{link}}$ multicast blocks. The corresponding total reward will be receiving $x_1\mathcal{D}_{\text{pay}}$ multicast blocks and eliminating $x_2\mathcal{D}_{\text{pay}}$ blocks of debts, where x_1 and x_2 are non-negative integers and $x_1 + x_2 = x$. Next because D's utility function is such that it benefits from sending one multicast block and then receiving σ multicast blocks, the reward exceeds the cost as long as $(x+\mathcal{D}_{\text{link}})\cdot\max(1,\sigma) < x\mathcal{D}_{\text{pay}}$. For sufficiently large x, this inequality is guaranteed by $\max(1,\sigma) < \mathcal{D}_{\text{pay}}$ as in Equation 1 in the theorem. Thus propagating multicast blocks (and issuing doins) using the debt-link will increase D's utility. Since the collusion strategy α is pareto-optimal, D must issue doins on the debt-link — otherwise α will be dominated by some "better" collusion strategy.

We have now proved that at the end of any round, each peer in the DCast execution gets at least those multicast blocks as it would get in the reference execution. We next would like to reason about the cost bits that a non-deviator sends/receives in the DCast execution. First consider those multicast bits received by this non-deviator, either directly from the root or pulled by this non-deviator from other peers (in which case the non-deviator will pay for or relay the corresponding doin later). Because α is pareto-optimal, Theorem 4 tells us that it must be non-damaging. By definition of non-damaging collusion strategies, we know that the number of cost bits that a non-deviator sends/receives in the DCast execution is at most $\mathcal{D}_{\text{root}}$ for each such multicast bit received (since $1 < \mathcal{D}_{\text{pay}} < \mathcal{D}_{\text{root}}$). Next, a non-deviator may incur further cost bits when it issues new doins. Under a non-damaging collusion strategy, every doin is paid. Thus if a non-deviator manages to issue new doins, it will receive $\mathcal{D}_{\text{pay}}$ multicast bits for each cost bit incurred, or equivalently, incur $1/\mathcal{D}_{\text{pay}}$ cost bit for each multicast bit received. Since $1/\mathcal{D}_{\text{pay}} < 1 < \mathcal{D}_{\text{pay}} < \mathcal{D}_{\text{root}}$, this can only increase the utility of the peer above the safety-net utility. □